Language Support for the Specification and Development of Composite Systems

MARTIN S. FEATHER
USC/Information Sciences Institute

When a complex system is to be realized as a combination of interacting components, development of those components should commence from a specification of the behavior required of the composite system. A separate specification should be used to describe the decomposition of that system into components. The first phase of implementation from a specification in this style is the derivation of the individual component behaviors implied by these specifications.

The virtues of this approach to specification are expounded, and specification language features that are supportive of it are presented. It is shown how these are incorporated in the specification language Gist, which our group has developed. These issues are illustrated in a development of a controller for elevators serving passengers in a multistory building.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*languages*; *methodologies*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics*; D.3.2 [**Programming Languages**]: Language Classifications—*very high-level languages*; D.3.3 [**Programming Languages**]: Language Constructs; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*specification techniques*

General Terms: Design, Languages, Theory, Verification

Additional Key Words and Phrases: Composite systems, distributed decomposition, interactive systems

1. INTRODUCTION

There is a growing consensus that, in order to achieve major improvement in software production and maintenance, the entire programming process must be formalized and given machine support (see, for example, the joint report of several researchers [7]). The keystone of such an approach is the formal specification of the requirements of the task to be programmed. Our specification group

This research was supported in part by the NSF under contract MCS-7918792, and in part by the Defense Advanced Research Projects Agency under contract MDA903 81 C 0335. Views and conclusions contained in this document are those of the author and should not be interpreted as representing the official opinion or policy of DARPA, the NSF, the U.S. Government, or any other person or agency connected with them.

Author's address: USC/Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90292.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1987 ACM 0164-0925/87/0400-0198 $00.75

at ISI[1] has pursued this philosophy in developing a specification language and companion techniques to support the transformational derivation of implementations from specifications. Our views and techniques are elaborated and defended elsewhere [1].

The focus here is the specification and development of *composite* systems, by which I mean systems whose realization will be multiple interacting components. The software development task may be to implement one or several of these components. The remaining components of the system form the implementation's environment, and might consist of other software systems, physical machinery, human beings, and so on.

A specification should be a lucid description of a task or activity. Its lucid nature derives from an emphasis on a natural description of the task, without regard for implementation concerns. Hence, when a complex system is to be realized as a combination of interacting components, development of those components should commence from a specification of the whole system. To be natural and lucid, such a specification should be of the behavior required of that composite system. The decomposition of that system into components should be specified separately. These specifications disregard the *implementation* concern of how to allocate behaviors to each of the components so that their combination will realize the composite system.

We call this style of specification *closed-system specification*, to emphasize that the specification of the composite system is self-contained, that is, closed, in that there is no interaction with anything outside of the specification. In contrast, a specification that has an interface to some unspecified environment would be "open."

The ensuing sections present the following:

—The benefits of the closed-system style of specification and its role in the software development process.
—Some specification language features that are supportive of the closed-system style of specification; these features have been embodied in our specification language, Gist.
—Additional specification language features that formalize the intuitive notions of "choice" and "responsibility," of use in expressing closed-system specifications and stages in developments from closed to open specifications. It is shown how Gist could be extended to include these features.
—A lengthy example to illustrate the above concepts put to use.
—Related research and conclusions.

2. ELABORATION OF THE CLOSED-SYSTEM STYLE OF SPECIFICATION

Specification of a composite system is divided into a specification of the behavior required of that composite system and a specification of how that system is to be decomposed into components. For example, consider the scenario of elevators

[1] Headed by B. Balzer, and currently including D. Cohen, M. Feather, N. Goldman, L. Johnson, B. Swartout, D. Wile, and K. Yue. Former members who have made significant contributions to this research are W. Chiu, L. Erman, S. Fickas, P. London, and J. Mostow.

serving passengers in a multistory building. The components are the individual passengers, the elevator controller, and the elevator mechanism. The behavior required of this composite system is the rapid transportation of passengers to their destination floors. Decomposition defines what interaction is allowed between these components, for example, the controller may be allowed to issue start/stop commands to the elevator motors, thus causing movement of elevators; passengers may be allowed to signal to the controller their presence at a floor and their desire to be transported in some direction.

The potential benefits of this style of specification and some of its implications for the subsequent implementation effort are discussed in the sections that follow.

2.1 Potential Benefits of Closed-System Specification

2.1.1 *Explicit Description of a Whole System.* Contrast the closed-system style of specification, in which the whole system is specified explicitly, to the open style, in which a single component interacts through some interface with an environment that is not explicitly specified. In the former, the full power and richness of the specification language can be used to describe the assumptions that the component to be implemented may make regarding its environment. In the latter, those assumptions must be expressed as part of the interface, the language for which is typically not the full specification language (e.g., it might be limited to applicability conditions on the component's externally invocable routines).

A closed-system specification may be used to explore the implications and ramifications of the system. Because the whole system is specified, it may serve as a self-contained prototype suitable for immediate testing. Testing might take the form of symbolic evaluation, simulation, property proving, or any combination of these. Just as specification benefits from the ability to describe system-wide behaviors, so these investigative methods are enhanced by the ability to use them to explore system-wide behaviors.

2.1.2 *Explicit Description of Decomposition.* When a system is decomposed into components, the components are usually restricted in the extent to which they may interact with one another. The decomposition determines what information "belongs to" a component and which activities are done by which components. It also determines the restrictions on interactions among components, that is, what information belonging to one component may be accessed or affected by an activity of another component. For example, within the elevator scenario, a passenger's destination is information "belonging to" that passenger. The controller is restricted from affecting that value or from accessing it.

The implementations of a system's components, when combined, must achieve the specified system behavior while complying with the restrictions on interactions among components.

The specification of a system's decomposition should be explicit and separate from the specification of that system's behavior. Each of these is a specification, and hence may be expressed without regard to the restrictions implied by the decomposition. For example, in specifying the behavior of elevators, part of the specification may be that an elevator containing passengers must move

in the direction of the destinations of those passengers—this system behavior is expressed in terms of elevator locations and passenger locations and destinations, without regard to the way the elevator component is restricted in its access to knowledge of passengers and vice versa.

2.1.3 *Component Behaviors Derived from System Behavior and Decomposition.* An especially powerful consequence of the separate and explicit specifications of overall system behavior and system decomposition is that they serve as the starting point from which to *derive* the behaviors required of the components. Contrasting this with an open specification of a component, we see that the latter must have decomposed the composite system behavior as a prerequisite to expressing it.

By deriving component behavior from specifications of system behavior and decomposition, the derivation process itself can be formally recorded and supported. Further, it retains maximum freedom of choice among alternative decompositions of system behavior into component behaviors. In contrast, starting the development from an already decomposed system leaves a larger gap between what is formally specified (individual component behaviors) and what is the unformalized intent (system behavior). Also, in order to express the individual component behaviors, one out of possibly many decompositions of the system behavior must already have been selected, thus prematurely constraining the options for implementing that component.

2.2 Development—Closed-System Specifications to Implementations of Components

2.2.1 *Development Process.* The closed-system specification of a composite system, together with a specification of its decomposition into components, serves as the starting point for developing implementations of one or more of those components. Since the closed-system style implicitly defines component behavior, a major goal of such a development will be to derive an explicit definition from this. In all but the most trivial of examples, this will not be accomplished in a single step, but rather will involve the gradual decomposition of system-wide behavior into individual behaviors allocated to the components. The result will be an explicit specification of the behavior required of an individual component, in the open style of specification. This paper focuses on language features that facilitate expression of closed-system specifications and intermediate stages toward open-style specifications of their components.

2.2.2 *Intertwining of Specification and Implementation.* Swartout and Balzer [14] argue that specification and implementation are strongly intertwined. They suggest that we should not expect to develop software by first constructing a specification without any consideration of resource limitations, and thereafter transforming that specification to introduce efficiency while completely preserving its functional behavior. Rather, the ideals of the specification will undergo multiple modifications as implementation reveals necessary compromises. This blurs the distinction between specification and implementation. We infer that, to achieve our goal of offering automated support to software production and maintenance, we must record and support this intertwining as part of the development process.

Developments from composite system specifications give rise to new forms of intertwining. The ideals of a composite system specification are the behaviors required of the overall system and the decomposition of that system into components. In realizing such a system as interacting components, it may be necessary to modify these ideals as a result of interactions discovered during the development process. For example, in order to coordinate activities in different components it may be necessary to extend the components' interfaces (defined in the decomposition) to allow more communication between them. Alternatively, the interface might be left unchanged, and instead the overall system behavior extended to include the behaviors that result from combining uncoordinated components.

3. A LANGUAGE FOR CLOSED-SYSTEM SPECIFICATION

Our specification group at ISI has developed a specification language, *Gist*. The origin of this language is Balzer and Goldman's study of the principles of good software specification [2]. Three of their proposed principles are that

—the specification must encompass the system, of which software is a component;
—the specification must encompass the environment in which the system operates; and
—the specification must be operational.

These principles are clearly pertinent to the closed-system style of specification; and hence in constructing Gist to satisfy these principles, we believe we have emerged with a language supportive of such a style. Of course, Gist's language features that provide this support could probably be used with other specification languages without too much difficulty.

A comprehensive description of Gist is beyond the scope of this paper; instead, I will present only a summary of the language, and detail only those parts which are of immediate relevance to closed-system specification. For a fuller description of both the language and the motivations that shaped its design, see [3].

3.1 Denotation of a Gist Specification

The overall meaning of a Gist specification of some application is as follows:

> The specification denotes the set of acceptable histories that the application may exhibit. Each history comprises an initial state and a sequence of transitions. The transitions correspond to activity in the application domain; applying a sequence of transitions to the initial state yields the state corresponding to the application domain after those transitions have occurred. States model the application domain at instants in time by means of objects and associations among those objects; transitions modify the existence of objects and associations among them.

The transition structure of each "history" facilitates the expression of dynamic activity taking place over time. This structure permits the direct modeling of applications such as process control, communication systems, and operating systems, where the system activity is not easily characterized as a function mapping inputs to outputs, but is an ongoing series of interdependent interactions

that affect the underlying domain. A specification denotes a *set* of such histories to represent the possibly many alternative histories that the system may exhibit. The "objects and associations among those objects" comprise Gist's data-modeling capability, which is essentially a typed entity-relationship model. This model is appropriate for specification because of its neutrality toward any particular representation that might be chosen for implementation.

This paper is not concerned with the details of Gist's data model (namely, what comprises states and what are legal transitions between states), so some simplifying abstractions and assumptions are made. A transition is regarded as simply a set[2] of primitive changes called *deltas*, where the only property of deltas is that there is an equality relation between them. It is assumed that all histories begin in the same initial state.[3]

The simplified denotation is thus

a Gist specification denotes a *behavior*, which is a set of *histories*;
a *history* is a sequence of *transitions*;
a *transition* is a set of *deltas*;
a *delta* is primitive; there is an equality relation over deltas.

3.2 Determining the Denotation—Generation and Pruning

A Gist specification comprises a *generative* component, which denotes a set of candidate histories, and *constraints*, which denote predicates on histories. The denotation of the specification is the set of all those candidate histories that satisfy all the constraints.

In practice, this generate-and-prune paradigm is applied as follows: the generative portion straightforwardly defines a set of histories, encompassing all the desired ones; the constraints separately specify characteristics and requirements not necessarily ensured by the generative portion. The resulting denotation consists of precisely those generated histories that satisfy all the constraints. This style is not new to specification; Darlington, in his transformational derivation of sorting algorithms [5], begins with a specification of sorting that generates all permutations of the input list, and thereafter filters to retain only those permutations that are ordered. Where Gist makes a major divergence from other specification languages is in incorporating this paradigm as fundamental to determining denotation. Further, because Gist denotations are sets of histories, constraints may refer to information spread through a history, making easy the definition of constraints that refer to activities taking place over time (for example, that the salary of an employee must be monotonic and increasing; that, having touched a chesspiece, a player must thereafter move that piece; that messages must be received in the order in which they were sent).

In a specification of the composite system behavior, both the generative portion and the constraints may be defined in terms of information gathered from

[2] Gist thus has *merging* semantics for parallelism, allowing a single transition to be composed of several deltas. This permits the natural modeling of simultaneous activity, akin to "atomic transitions" in databases.

[3] If necessary, introduce an initial empty state and appropriate transitions to lead to each of the intended starting states.

throughout the specified system. In other words, system decomposition in no way limits expression of system behavior. As we shall see, in modeling the stages that occur in developments from closed- to open-system specifications, it is important to make generation and pruning sensitive to the decomposition.

3.2.1 *Generation of Candidate Histories.* The generative portion of a Gist specification is built from primitive statements that cause state changes, combined using conventional control constructs to provide the following:

sequentiality	stop the elevator; open the elevator doors;
conditionality	*if* there is a passenger on board *then* . . .
iteration	*for all* floors at which there are waiting passengers *do* . . .
choice	*choose* open the elevator doors *or* move on to the next floor
parallelism	open the elevator doors *and* open the elevator shaft doors
procedural abstraction	*call* SEND-ELEVATOR-TO-FLOOR[elevator2, floor5] *where* SEND-ELEVATOR-TO-FLOOR[*e* of type elevator, *f* of type floor] = . . .
data-driven invocation (demons)	*whenever* the elevator arrives at a floor *do* . . .

In defining a closed-system specification, the language of predicates and expressions used by these control constructs has access to system-wide information (e.g., the location of elevators, the status of their doors, or the destinations of passengers). Also, the language has access to previous and future states within the entire sequence of states that comprise a history (e.g., the last floor this elevator stopped at, the next floor this elevator will stop at).

These liberal access abilities lend Gist considerable power of expression not found in other specification languages.

3.2.2 *Constraints and Pruning.* For convenience, a constraint is usually expressed as a predicate to be evaluated inside a state, in which case it will be true of a history if and only if that predicate evaluates to true in every state in the history. For example, the constraint that every passenger must have exactly one destination floor will be true of a history if and only if in every state of that history, every passenger in that state has exactly one destination floor. The predicate defining the constraint may of course be expressed in terms of information drawn from throughout the system, and may refer to previous and future states of the history.

In what follows, it will not matter how a constraint is defined, only whether or not a history satisfies a constraint. Hence a constraint will be represented as a predicate on histories:

$$\text{constraint: history} \rightarrow \text{boolean}$$

If c is a constraint and h is a history, write $c(h)$ to denote the constraint applied to the history, that is, $c(h)$ will be true if the history satisfies the constraint, false otherwise.

Pruning a behavior with a set of constraints retains those and only those histories of the behavior that satisfy all the constraints. This is formalized in function Prune, which, given a behavior b and a set of constraints sc, returns the pruned behavior.

Definition. Prune

$$\text{Prune: behavior} \times \text{set of constraints} \rightarrow \text{behavior}$$

$$\text{Prune}(b, sc) = \{h \in b \mid (\forall c \in sc \mid c(h))\}$$

3.2.3 *Some Properties of Pruning Constraints.* The following properties follow immediately from the definition of constraints and pruning; we shall see later that, as we extend their definitions to take into account the decomposition into components, not all of these properties are preserved.

Uniqueness. For any behavior b and set of constraints sc, Prune(b, sc) uniquely defines a behavior.

Thus a specification denotes a uniquely defined behavior. Note that since a behavior is a set of histories, nondeterminism is modeled as multiple histories within that set. An implementation of a specification will be correct with respect to behavior, provided the implementation's behavior is a (nonempty) subset of the specification's behavior.

Monotonic. For any behavior b, set of constraints sc, and constraint c,

$$\text{Prune}(b, \{c\} \cup sc) \subseteq \text{Prune}(b, sc).$$

Thus the addition of a constraint may cause more histories to be pruned out, but will never cause more histories to be retained.

Commutative. For any behavior b and sets of constraints $sc1$ and $sc2$,

$$\text{Prune}(\text{Prune}(b, sc1), sc2) = \text{Prune}(b, sc1 \cup sc2) = \text{Prune}(\text{Prune}(b, sc2), sc1).$$

Thus a calculation of pruning may be done incrementally, and the order in which the constraints are considered is irrelevant. In fact,

$$\text{Prune}(\text{Prune}(b, sc1 \cup sc2) = \text{Prune}(b, sc1) \cap \text{Prune}(b, sc2).$$

Thus pruning may be done independently for any factoring of the constraints and the results combined to give the same net result.

Conjunctive. For any behavior b, constraints $c1$ and $c2$, and set of constraints sc,

$$\text{Prune}(b, \{c1\} \cup \{c2\} \cup sc) = \text{Prune}(b, \{c3\} \cup sc) \quad \text{where } c3(h) = c1(h) \wedge c2(h).$$

Thus constraints may be conjoined, or conjunctive constraints decomposed, without changing the behavior.

3.3 Defining the Decomposition of a Closed-System Specification

3.3.1 *Components and Agents.* Composite systems are made up of *components*. Each autonomous process in the domain being specified will typically be a separate component (e.g., in a domain of elevators serving passengers in a

multistory building, each passenger, the elevator controller, and the elevator mechanism itself might be components).

Gist has a construct called an *agent*, which is used to model components. Agents partition the generative portion of a Gist specification. Thus every primitive statement (one that contributes a delta or deltas to a transition) is within the scope of some agent. When such a statement is executed, its containing agent is said to have "done" the deltas contributed by that statement. For example, if a statement within the passenger agent changes the passenger's location, that delta is said to have been done by that passenger agent. The information of which agent has done a delta is recorded within the denotation by labelling each delta with the agent that did it. To permit this record necessitates extending the definition of the denotation slightly, modifying a transition to be

> A *transition* is a set of ordered pairs, each of which is an *agent* × *delta*.
> An *agent* is primitive; there is an equality relation over agents.

The augmented denotation distinguishes between the same delta done by different agents within a single transition (whereas the unaugmented denotation would not). The language of predicates and expressions over histories (which is used in defining both the generation of candidate histories and constraints to be applied in pruning) may use this extra knowledge.

3.3.2 *Agent Interfaces.* I now propose an extension to Gist's agents, to model more of the aspects of components. The objective is to represent the restrictions on interactions between components. To do this requires the notion of information "belonging to" a component.

In addition to containing control statements, an agent may also contain declarative statements of the data model's objects and associations. Instances of such objects and associations are then said to "belong to" that agent.

An agent's implementation must *comply* with the restrictions on the information it accesses and affects. The default restrictions are that an agent may access and affect only the information that "belongs to" it. For example, if a passenger's destination is information belonging to the passenger agent, then the default restriction is that only the passenger agent may access or affect that information. These default restrictions may be overridden in agents' *interfaces*. For example, suppose an elevator's location "belongs to" the elevator mechanism agent. To model the fact that a passenger at a floor can see an open-doored elevator at that floor (and hence knows its location), the interface of the elevator mechanism grants the passenger agent the right to access that information under those circumstances. The precise means of expressing interfaces is still in tentative form; further experience is required to judge what will be the most generally useful syntactic notations, defaults, and so on.

Compliance with a restriction on affecting information means that when a delta that changes the information is done by an agent, that agent must be allowed to affect that information. For example, if the elevator mechanism were restricted from affecting a passenger's destination, it would not be permitted to change that value (otherwise it would be trivial to get a passenger to his/her destination: simply change his/her destination to be his/her current location!).

Compliance with a restriction on accessing information is not usually determinable until quite late in the development from specification to implementation. Typically, Gist specifications define their generative portions and constraints in terms of arbitrary access to information, without such access being recorded in the denotation. For example, the specification of the constraint that limits the movement of an elevator to the direction of its passengers' destinations might be expressed in terms of passengers' presence inside elevators, and passengers' destinations; but all that appears in the denotation is the movement of the elevator, without any representation of access to information about passengers. Ultimately, development will lead to an algorithm for controlling movement, at which point that algorithm's accesses can be made into explicit activities appearing in the denotation (e.g., sensing when buttons have been pushed—by passengers requesting transportation to particular floors). Only then will it be possible to check for compliance with access restrictions. Obviously, consideration of these restrictions may have an early influence on the implementation process.

4. CHOICE AND RESPONSIBILITY

The previous section outlined those features of Gist that are supportive of the closed-system style of specification. Now I show how Gist may be extended to formalize the intuitive notions of "choice" and "responsibility," notions which are useful in both specification and development. I begin with the trivial but illustrative example below.

4.1 Example Scenario—the Business Lunch

Consider a scenario of a business lunch with two participants, the host and the visitor. They eat at a restaurant with a very limited menu, offering only two choices: an expensive steak special, or a more moderate chef's salad. The host's company has an austerity measure to discourage overly large expense claims, and will only recompense the expense of a lunch if the total cost does not exceed some preset limit (otherwise no compensation whatsoever is given). As it happens, the cost of two steak specials exceeds this limit, whereas the cost of two chef's salads, or one chef's salad and one steak special, falls below this limit. Hence the two diners, if they wish to be compensated for their meal, must not both order steak. Suppose that the host orders first. This scenario is expressed in terms of behavior, histories, transitions, and so on, as follows.

The unpruned system behavior is the set of four histories, $\{h_1, h_2, h_3, h_4\}$ where

$h_1 = \{H \times \text{salad}\}; \{V \times \text{salad}\}$	The host orders salad, then the visitor orders salad.
$h_2 = \{H \times \text{salad}\}; \{V \times \text{steak}\}$	The host orders salad, then the visitor orders steak,
$h_3 = \{H \times \text{steak}\}; \{V \times \text{salad}\}$	and so on.
$h_4 = \{H \times \text{steak}\}; \{V \times \text{steak}\}$	

Notation. "H" (host) and "V" (visitor) are the agents in this system.
"H × salad" is the delta of ordering a salad, done by agent H (the host).
"{H × salad}" is the transition consisting of a single delta.
";" separates transitions in a history.

The limit on their expenditure may be expressed as a constraint which holds for histories in which they do not both order steak. Calling this constraint "NotBothSteak", we have

$$\begin{aligned} \text{NotBothSteak}(h_i) &= \text{true} \quad \text{for } i = 1, 2, 3 \\ &= \text{false} \quad \text{for } i = 4 \end{aligned}$$

Then

$$\text{Prune}(\{h_1, h_2, h_3, h_4\}, \{\text{NotBothSteak}\}) = \{h_1, h_2, h_3\}.$$

Intuitively, this system's behavior may be viewed as a sequence of *choices*, H's choice of lunch followed by V's choice of lunch. This may be seen by drawing the set of histories as a tree whose root is the common starting point of each history, and whose branches are the transitions; see Figure 1. Each path through the tree from root (the common starting point) to leaf (an end-point) is a history. Histories with common initial sequences of transitions share the same sequence of branches from the top of the tree, so that branch points within the tree occur when and only when histories diverge.

Viewed in this manner, it is clear that pruning leaves H's choice unrestricted, but constrains V's choice if H has chosen steak. Is this fair? To answer requires a notion of *responsibility*. Intuitively, those and only those agents *responsible* for a constraint should limit their choices to ensure satisfaction of that constraint. Thus, if V is responsible for the NotBothSteak constraint, then this *is* fair, because only V's choices are constrained. Conversely, if H instead of V is responsible, this is *not* fair. Instead, H's choice should be constrained to salad, leaving V's choice unconstrained: see Figure 2.

4.2 Formalizing Choice and Responsibility

The original simple form of pruning simply discards those candidate histories that fail any of the constraints, and so is neutral with respect to agents. This neutrality is appropriate when specifying the ideal behavior of a composite system. As such a system is decomposed into components, the necessity to decompose ideal system behavior into a composition of individual component behaviors introduces the need for discrimination among agents, and the notions of choice and responsibility play an important role in these descriptions.

The simple business lunch example illustrates the overall approach to formalizing the notions of choice and responsibility. The behavior, a set of histories, is viewed as a tree of histories, in which diverging branches in the tree correspond to diverging histories. At a point in the tree where branches diverge, an agent has the choice of the deltas it contributes to the transitions starting each of those branches. Pruning discards histories in order to satisfy the constraints; viewed on the tree, pruning lops off branches, and hence potentially limits agents' choices. Pruning must ensure that all the constraints are satisfied, and must do so by constraining the choices of only those agents responsible for the constraints.

4.2.1 *Combining the Set of Histories into a Tree.* A set of histories is combined into a rooted, unordered tree by maximally sharing common initial sequences of transitions. Thus each path through the tree from the root to a leaf corresponds

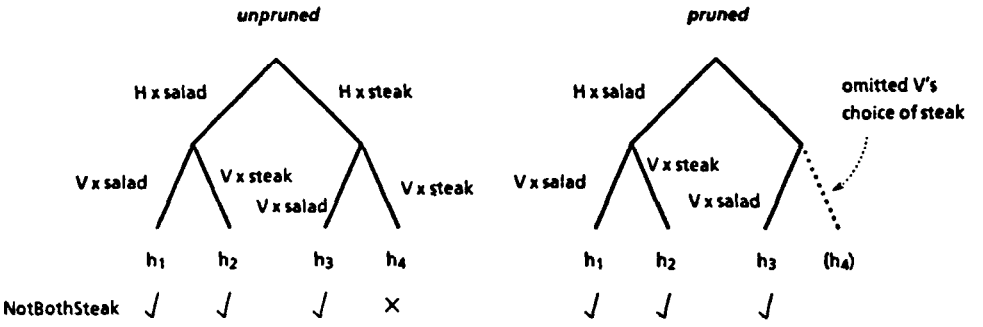


Fig. 1. The diners' behavior tree—unpruned and pruned.

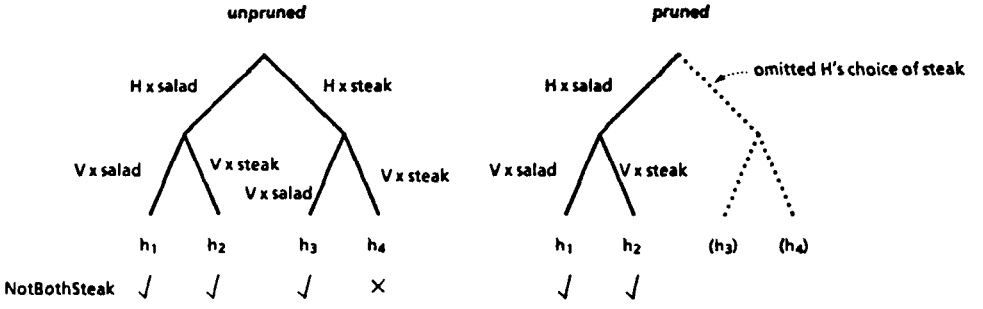


Fig. 2. The diners' behavior tree—pruned when H is responsible.

to one of the histories in the original set. This tree representation is more intuitive for understanding the notions of choice and responsibility.

The tree representation is interchangeable with the set representation, provided that no history in the behavior is an initial sequence of any other history. This restricts the behaviors that can be represented. Fortunately, the restriction is desirable, because if a behavior includes two histories, one of which is an initial sequence of the other, and the system performs the sequence of transitions that comprise the shorter history, then it would be impossible to tell whether the system had actually completed the shorter history or was at any moment about to continue with the longer one.

Our tree representation of behavior has much in common with Milner's CCS (Calculus of Communicating Systems) trees [8, 13]. In CCS, a program's behavior is determined by how it communicates with an observer (its environment); rooted, unordered trees with labeled arcs represent behaviors. CCS formalisms are suited to representing and reasoning about behaviors of what I am calling components and combinations of components.

4.2.2 *Choice.* An agent's choices at a node in a behavior tree are the different sets of deltas it contributes to the transitions emerging from that node. See, for example, Figure 3, the diners' pruned behavior tree.

Formally, define a function ChoicesAtNode which, given a node n in behavior tree b and agent a, returns that agent's choices (a set of sets of deltas) at that node.

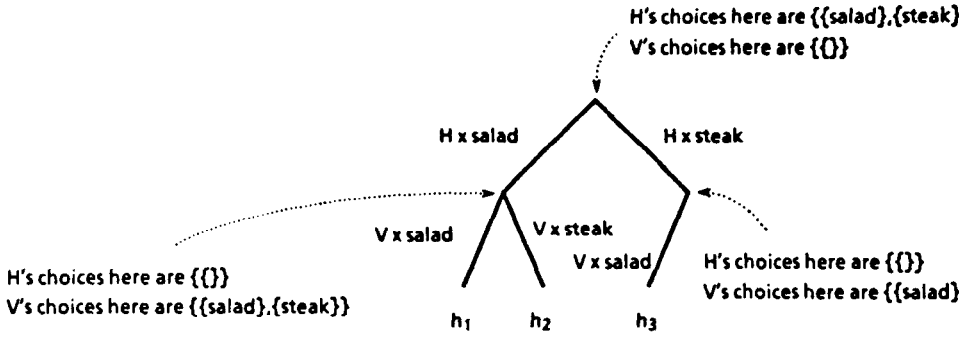


Fig. 3. Choices at nodes in the diners' pruned behavior tree.

Definition. ChoicesAtNode

ChoicesAtNode: node × behavior × agent → set of sets of deltas

$$\text{ChoicesAtNode}(n, b, a) = \{\{\delta \mid (a \times \delta) \in t\} \mid t \in \text{TransitionsFromNode}(n, b)\}$$

TransitionsFromNode(n, b) = the set of transitions that emerge from node n in behavior tree b (the empty set if n is not a node of b)

Pruning discards histories from the set of candidate histories. Viewed on the equivalent tree representation, pruning "lops off" branches of the tree. When a branch is lopped off, the entire subtree it supported is discarded; see Figure 4.

Define function LoppedAtNode which, given node n in behavior tree b and pruned behavior tree pb, returns the set of transitions beginning those branches lopped from node n in that pruning.

Definition. LoppedAtNode

LoppedAtNode: node × behavior × behavior → set of transitions

$$\text{LoppedAtNode}(n, b, pb) = \begin{cases} \text{TransitionsFromNode}(n, b) \\ - \text{TransitionsFromNode}(n, pb) \end{cases}$$

At nodes from which branches have been lopped, agents may have fewer choices in the pruned tree than they had in the unpruned tree (i.e., their choices may be *limited* by the pruning; see Figure 5).

Formally, define LimitedByLopping to compute those agents whose choices are limited in this manner when the branch beginning with transition t is lopped off from node n as part of pruning to behavior tree pb (i.e., those agents whose choice in the lopped transition is not among their choices remaining in the pruned tree pb).

Definition. LimitedByLopping

LimitedByLopping: transition × node × behavior → set of agents

$$\text{LimitedByLopping}(t, n, pb) = \{a \mid \{\delta \mid (a \times \delta) \in t\} \notin \text{ChoicesAtNode}(n, pb, a)\}$$

4.2.3 *Responsibility.* The agents *responsible* for a constraint limit their choices so as to ensure the satisfaction of the constraint without making it necessary for

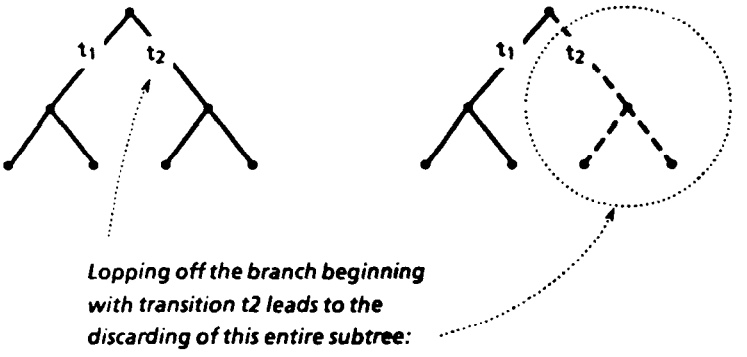
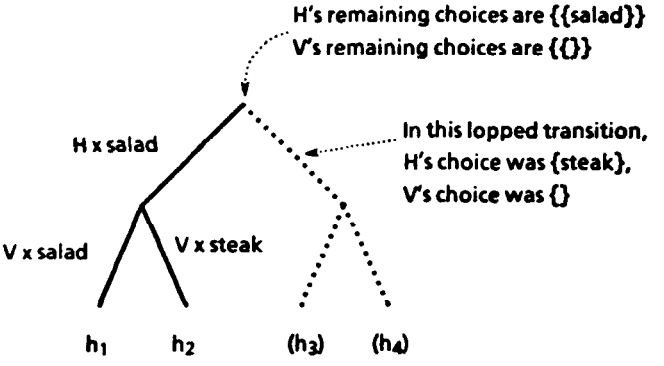


Fig. 4. Pruning lops off branches and their subtrees.



Since H's choice in the lopped transition is *not* among H's choices remaining in the pruned tree, this lopping *limits* H.
Conversely, since V's choice in the lopped transition *is* among V's choices remaining in the pruned tree, this lopping does *not* limit V.

Fig. 5. Limiting of agents' choices owing to lopping of branches during pruning.

other, nonresponsible, agents to limit their choices. Thus, if pruning which lops off a branch at a node has the effect of limiting the choices of some agents, those agents must be *responsible* for lopping off that branch. The following definitions determine which are the responsible agents.

Responsibility for a constraint is assigned to a set of agents. This is represented by pairing the set of responsible agents with the constraint, to form an *assigned constraint*:

$$\textit{assigned constraint} = \text{set of agents} \times \text{constraint}$$

When a history fails a constraint to which a set of agents is assigned, that set of agents is responsible for pruning out the history. When a history fails several such constraints, the set of responsible agents is the union of the sets of agents responsible for each of the failed histories. This is expressed in function ResponsibleForHistory which, given a history and a set of assigned constraints, returns the agents responsible for pruning out that history.

Definition. ResponsibleForHistory

ResponsibleForHistory: history × set of assigned constraints → set of agents

$$\text{ResponsibleForHistory}(h, sac) = \bigcup_{(sa \times c) \in sac \mid \sim c(h)} sa$$

At a node where pruning lops off a branch, the set of agents responsible for lopping that branch is the union of the agents responsible for the histories comprising that lopped-off branch. If a branch in a tree is identified by the transition beginning that branch, then the histories comprising that branch are those and only those containing that transition. In the definition that follows, "h contains t" means history h contains transition t. Formally, define ResponsibleForLopping(t, b, sac) to return the set of agents responsible for lopping off the branch beginning with transition t from behavior tree b when pruning for set of assigned constraints sac thus:

Definition. ResponsibleForLopping

ResponsibleForLopping: transition × behavior
× set of assigned constraints → set of agents

$$\text{ResponsibleForLopping}(t, b, sac) = \bigcup_{h \in b \mid h \text{ contains } t} \text{ResponsibleForHistory}(h, sac)$$

These simple definitions result in a rather crude assignment of responsibility when a history fails several constraints at once, or when several histories pruned by the lopping of the same branch fail several constraints (in either case, the sets of agents responsible for the failed constraints are simply unioned together to determine the responsible agents for pruning). For example, as a consequence, it is impossible to differentiate between two agents being separately responsible for the same constraint and those two agents being jointly responsible for that constraint. Whether a more sophisticated definition is required remains to be seen. For the time being, the simple version set forth above will suffice.

4.2.4 *Pruning for Assigned Constraints.* Define predicate OKPruning(b, pb, sac) to be true if, at every node retained in the pruned behavior tree pb obtained by pruning b with set of assigned constraints sac, the agents whose choices are limited by lopping a branch from that node are a subset of the agents responsible for lopping that branch.

Definition. OKPruning

OKPruning: behavior × behavior × set of assigned constraints → boolean

$$\text{OKPruning}(b, pb, sac) = \forall n \in pb \mid \forall t \in \text{LoppedAtNode}(n, b, pb) \mid \textit{LimitedByLopping}(t, n, pb) \subseteq \text{ResponsibleForLopping}(t, b, sac)$$

Implicit within this definition is the assumption that nodes *not* retained within the pruned behavior tree need not be considered. Intuitively, if a choice point (node) is never reached, it does not matter what choices used to be there (i.e., one way to avoid making nonresponsible agents limit their choices is to not let them get to the state where they would choose).

Also implicit in this definition is the decision to consider the validity of lopping at a node on a branch-by-branch basis. Alternative definitions, which determine

the validity of lopping at a node by simultaneously considering all the lopped-off branches at once, have to be constructed with great care to retain the desirable closure property (that the union of two prunings OK with respect to choice and responsibility is also OK). Again, the simple definition set forth above is chosen here.

Pruning is achieved by discarding histories from the set of candidate histories; the predicate OKPruning defines when this is done in compliance with the notions of "responsibility" and "limiting of choice." There remains the original purpose of pruning, namely the discarding of every history that fails any constraint. These conditions are embodied in function OKPrunings(*b*, *sac*), which for behavior *b* and set of assigned constraints *sac* returns the set of all acceptably pruned behaviors.

Definition. OKPrunings

OKPrunings: behavior × set of assigned constraints → set of behaviors

$$\text{OKPrunings}(b, sac) = \{pb \mid pb \subseteq b \wedge \text{OKPruning}(b, pb, sac) \wedge (\forall h \in pb, (sa \times c) \in sac \mid c(h))\}$$

OKPrunings returns a *set* of behaviors, because for a given behavior and set of assigned constraints there may be several prunings that satisfy the requirements. The empty set of histories is always one of these behaviors. Also, OKPrunings is closed under union,[4] that is,

$$\forall b, sac, pb1 \in \text{OKPrunings}(b, sac), pb2 \in \text{OKPrunings}(b, sac) \mid (pb1 \cup pb2) \in \text{OKPrunings}(b, sac)$$

This implies that a behavior never has two "incompatible" prunings, neither of which can be extended to include the other. Also, for all finite sets of histories, this guarantees that OKPrunings is uniquely defined (certain pathological infinite sets of histories do not have a uniquely defined largest pruning).

These properties imply that, for a behavior that is a *finite* set of histories, applying OKPrunings yields a lattice of behaviors, where the links of the lattice correspond to subset. The (uniquely defined) largest behavior is the desired result of pruning, since it discards as few of the candidate histories as necessary. PruneA is defined to return this result, given behavior *b* and set of assigned constraints *sac*.

Definition. PruneA

PruneA: behavior × set of assigned constraints → behavior

$$\text{PruneA}(b, sac) = \text{Largest}(\text{OKPrunings}(b, sac)) = \bigcup_{bs \in \text{OKPrunings}(b, sac)} bs$$

4.3 Examples and Properties of Pruning Assigned Constraints

Pruning of unassigned constraints is simple in that the decision to retain or discard a candidate history is not dependent upon the presence or absence of other histories in the set of candidate histories. In constrast, pruning of assigned

[4] See the appendix for a sketch of a proof of this.

constraints *is* dependent upon the whole set of candidate histories, since it is that set which defines agents' choices. As a result, only some of the properties of unassigned pruning mentioned in Section 3.2.3 remain properties of assigned pruning, as is shown next.

Returning to the scenario of a business lunch.

The unpruned behavior is

$$\{h_1, h_2, h_3, h_4\} \quad \text{where } h_1 = \{H \times \text{salad}\}; \{V \times \text{salad}\}$$
$$h_2 = \{H \times \text{salad}\}; \{V \times \text{steak}\}$$
$$h_3 = \{H \times \text{steak}\}; \{V \times \text{salad}\}$$
$$h_4 = \{H \times \text{steak}\}; \{V \times \text{steak}\}$$

$$\text{NotBothSteak}(h_i) = \text{true for } i = 1, 2, 3$$
$$= \text{false for } i = 4$$

Pruning of assigned constraints gives

$$\text{OKPrunings}(\{h_1, h_2, h_3, h_4\}, \{\{H, V\} \times \text{NotBothSteak}\})$$
$$= \{\{h_1, h_2, h_3\}, \{h_1, h_2\}, \{\}\}$$

(see Figure 6), hence

$$\text{PruneA}(\{h_1, h_2, h_3, h_4\}, \{\{H, V\} \times \text{NotBothSteak}\}) = \{h_1, h_2, h_3\}$$

Extending the example, introduce the additional constraint that the visitor is a vegetarian (call this constraint "VegetarianV"):

$$\text{VegetarianV}(h_i) = \text{true for } i = 1, 3$$
$$= \text{false for } i = 2, 4 \qquad \text{See Figure 7.}$$

Then pruning of unassigned constraints gives

$$\text{Prune}(\{h_1, h_2, h_3, h_4\}, \{\text{NotBothSteak}\}) = \{h_1, h_2, h_3\}$$
$$\text{Prune}(\{h_1, h_2, h_3, h_4\}, \{\text{VegetarianV}\}) = \{h_1, h_3\}$$
$$\text{Prune}(\{h_1, h_2, h_3, h_4\}, \{\text{NotBothSteak, VegetarianV}\}) = \{h_1, h_3\}$$

4.3.1 *Properties of Pruning Assigned Constraints. Uniqueness.* Provided that the unpruned behavior is a finite set of histories, pruning of assigned constraints is guaranteed to uniquely define a behavior, that is, retains the important property of *uniqueness* of the denotation of specification.

Relationship to unassigned pruning. Note that, in the above example, the result of pruning with the NotBothSteak constraint assigned to all the agents in the system is identical to the result of pruning when that constraint was unassigned. This is true in general, that is, for any behavior b and set of constraints sc,

$$\text{Prune}(b, sc) = \text{PruneA}(b, sac)$$
$$\text{where } sac = \{(sa \times c) \mid c \in sc\}$$
$$\text{where } sa = \{a \mid \exists \delta, t, h \mid (a \times \delta) \in t \wedge t \in h \wedge h \in b\}$$

In the above, sa is the set of all agents contributing deltas to the system, that is, all those agents a such that, for some delta δ, the pair $(a \times \delta)$ is in a transition t contained in some history h of behavior b.

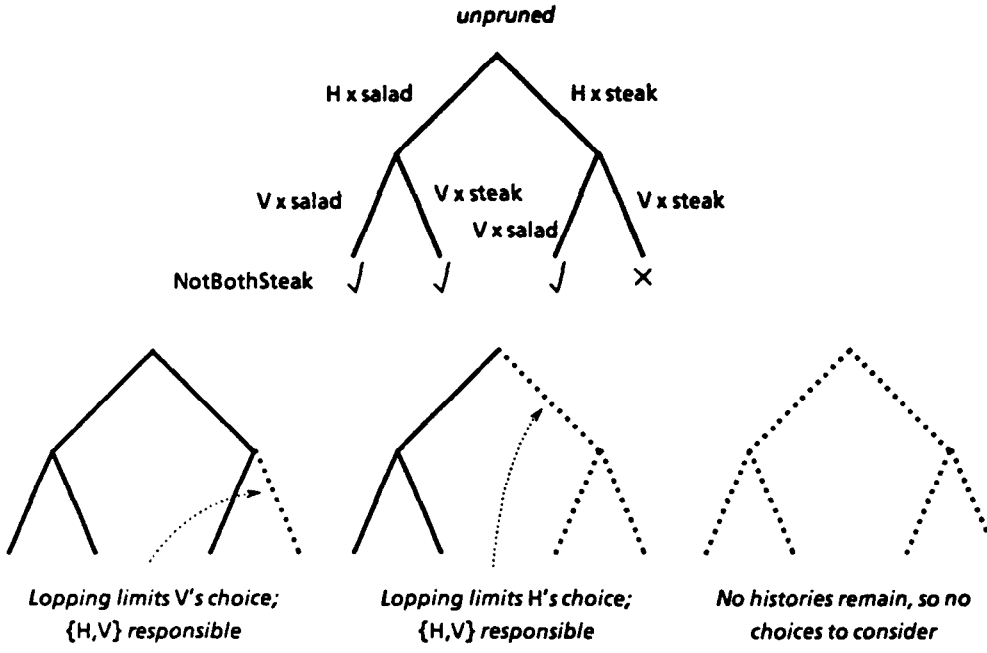


Fig. 6. OKPrunings of the NotBothSteak constraint assigned to both diners.

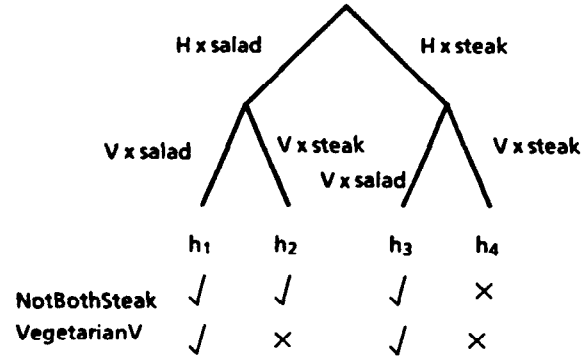


Fig. 7. The business lunch: unpruned behavior and constraints.

It follows immediately from the definition of PruneA (pruning of assigned constraints) that pruning a behavior with a set of assigned constraints is always a subset of pruning that behavior with those constraints unassigned. In other words, for any behavior b and set of assigned constraints sac,

$$\text{PruneA}(b, sac) \subseteq \text{Prune}(b, sc) \text{ where } sc = \{c \mid (sa \times c) \in sac\}$$

For example,

$$\text{PruneA}(\{h_1, h_2, h_3, h_4\}, \{\{H\} \times \text{NotBothSteak}\}) = \{h_1, h_2\}, \text{ and}$$
$$\text{Prune}(\{h_1, h_2, h_3, h_4\}, \{\text{NotBothSteak}\}) = \{h_1, h_2, h_3\}$$

At worst, the result of pruning may be the empty set of histories. For example, H cannot alone take the responsibility for the VegetarianV constraint, since either of H's choices leaves V free to choose between steak and salad; hence;

$$\text{PruneA}(\{h_1, h_2, h_3, h_4\}, \{\{H\} \times \text{VegetarianV}\}) = \{\ \}$$

Conjunctive. As in the unassigned case, constraints may be conjoined, or conjunctive constraints decomposed, without changing the effects of pruning. Thus, for any behavior *b*, set of agents *sa*, constraints *c1*, *c2*, and set of assigned constraints *sac*,

$$\text{PruneA}(b, \{sa \times c1\} \cup \{sa \times c2\} \cup sac) = \text{Prune}(b, \{sa \times c3\} \cup sac) \quad \text{where}(c1 \wedge c2)(h) = c1(h) \wedge c2(h)$$

For example,

$$\text{PruneA}(\{h_1, h_2, h_3, h_4\}, \{\{V\} \times \text{NotBothSteak}, \{V\} \times \text{VegetarianV}\}) = \{h_1, h_3\}$$

and

$$\text{PruneA}(\{h_1, h_2, h_3, h_4\}, \{\{V\} \times \text{NotBothSteak\&VegetarianV}\}) = \{h_1, h_3\}$$

$$\text{where NotBothSteak\&VegetarianV}(h) = \text{NotBothSteak}(h) \wedge \text{VegetarianV}(h)$$

In addition, pruning with the same constraint separately assigned to several sets of agents is identical to pruning with that constraint assigned to the union of those agent sets. Thus, for any behavior *b*, sets of agents, *sa1*, *sa2*, constraint *c*, and set of assigned constraints *sac*:

$$\text{PruneA}(b, \{sa1 \times c\} \cup \{sa2 \times c\} \cup sac) = \text{PruneA}(b, \{(sa1 \cup sa2) \times c\} \cup sac).$$

For example,

$$\text{PruneA}(\{h_1, h_2, h_3, h_4\}, \{\{H\} \times \text{NotBothSteak}, \{V\} \times \text{NotBothSteak}\}) = \{h_1, h_2, h_3\}$$

$$\text{PruneA}(\{h_1, h_2, h_3, h_4\}, \{\{H, V\} \times \text{NotBothSteak}\}) = \{h_1, h_2, h_3\}$$

Not necessarily monotonic. The behavior that results from pruning with a set of assigned constraints is *not necessarily* a subset of the behavior that results from pruning with a superset of those assigned constraints, that is, addition of another assigned constraint may cause some previously retained histories to be pruned, and/or some previously pruned histories to be retained. For example,

$$\text{PruneA}(\{h_1, h_2, h_3, h_4\}, \{\{H\} \times \text{NotBothSteak}\}) = \{h_1, h_2\}$$

Because H alone is responsible for pruning out h_4, it is H's choice that has to be limited, thus necessitating pruning out h_3 also.

$$\text{PruneA}(\{h_1, h_2, h_3, h_4\}, \{\{H\} \times \text{NotBothSteak}, \{V\} \times \text{VegetarianV}\}) = \{h_1, h_3\}$$

Adding the VegetarianV constraint assigned to V extends the set of agents responsible for pruning h_4 (which fails both constraints), hence V's choice may now be limited to effect pruning of h_4. This allows retention of h_3 (however, h_2 is now pruned out because it fails the introduced constraint).

This example shows that, when pruning simultaneously with several assigned constraints, these constraints do interact with one another. The very simple nature of pruning of unassigned constraints obscured any such interaction.

Not necessarily commutative. Multistage pruning of assigned constraints (in which the pruned behavior of one stage of pruning is used as the unpruned behavior for the next stage) is *not necessarily* equivalent to single-stage pruning, and the order in which the pruning is done *may* make a difference. For example,

$$\text{PruneA}(b, \{\{V\} \times \text{VegetarianV}\}) = \{h_1\}$$
$$\text{where} \quad b = \text{PruneA}(\{h_1, h_2, h_3, h_4\}, \{\{H\} \times \text{NotBothSteak}\})$$

Pruning first with $\{\{H\} \times \text{NotBothSteak}\}$ forces H's choice to be limited to salad, $\{h_1, h_2\}$, whereupon pruning with $\{\{V\} \times \text{VegetarianV}\}$ removes h_2.

$$\text{PruneA}(b, \{\{H\} \times \text{NotBothSteak}\}) = \{h_1, h_3\}$$
$$\text{where} \quad b = \text{PruneA}(\{h_1, h_2, h_3, h_4\}, \{\{H\} \times \text{VegetarianV}\})$$

Conversely, pruning first with $\{\{V\} \times \text{VegetarianV}\}$ forces V's choice to be limited to salad, $\{h_1, h_3\}$, whereupon pruning with $\{\{H\} \times \text{NotBothSteak}\}$ need remove nothing further; see Figure 8.

This phenomenon may be used to advantage, as is discussed in the following section. Successive stages in multistage pruning are monotonic, since the output of a stage of pruning is necessarily equal to or a subset of its input.

4.4 Reliance Among Constraints

A set of constraints may *rely* upon the effects of another set of constraints as follows: if we begin with a behavior (B1), prune first with a set (S1) of constraints to get another behavior (B2), and prune B2 with another set (S2) of constraints, then S2 *relies* upon the effects of S1's pruning because it has to deal only with the histories in B2, not with whatever additional histories were present in B1 and removed by the first pruning.

This phenomenon can be demonstrated in the business lunch scenario; see Figure 8.

When pruning with $\{H\} \times \text{NotBothSteak}$ follows pruning with $\{V\} \times \text{VegetarianV}$, H can rely upon V's pruning to have removed the history in which V chose steak. Hence H need prune no further, in particular, can retain choice of steak. Conversely, if the prunings are done in the other order, H cannot rely upon V's pruning, and so must prune out choice of steak. In this case, although V can rely upon H's pruning, there is no advantage to be gained.

In practice, it is quite useful to simplify the expression of constraints. For example, the specifier of an elevator controller may rely on passengers being unable to enter an elevator whose doors remain closed (something that might well be expressed by means of a constraint) in defining the constraint that prohibits passengers from ever moving further from their destinations. The latter may be expressed quite simply, without the need to be concerned with impossible transportation histories in which passengers enter closed elevators.

Note that when dealing purely with unassigned constraints, this phenomenon does not arise, because such constraints are not sensitive to choices.

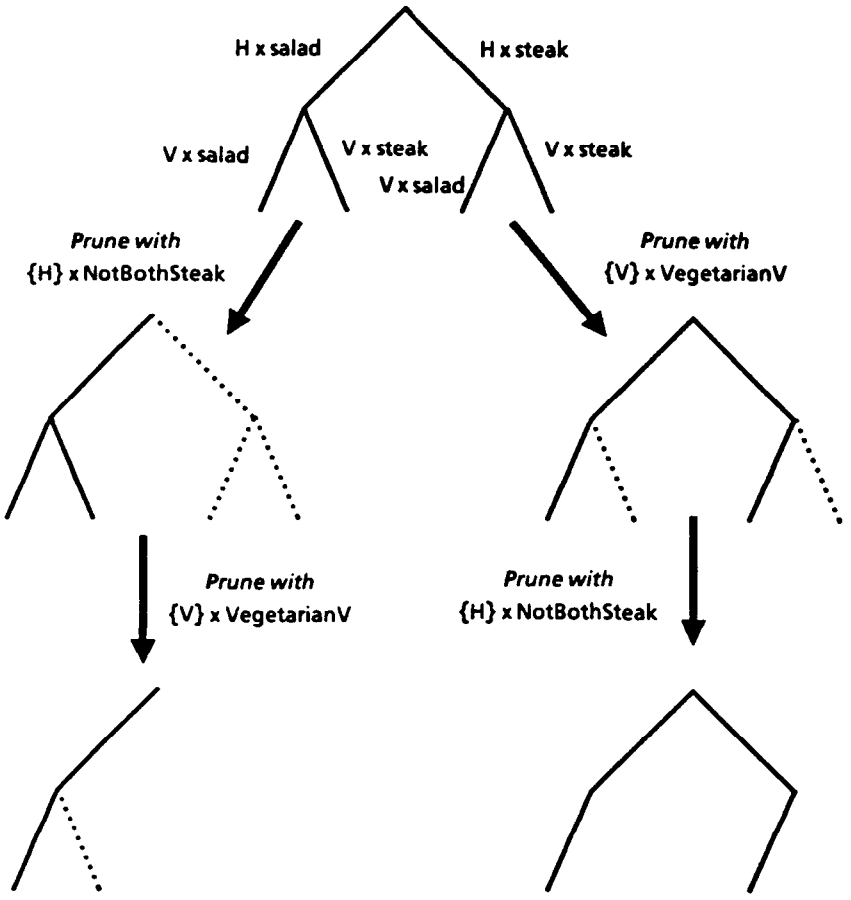


Fig. 8. Alternative multistage prunings in the business lunch scenario.

5. EXAMPLE—ELEVATOR SCENARIO

The scenario of controlling elevators in a multistory building is used to illustrate composite system specification and development from closed to open style, which is a necessary precursor to implementation.

5.1 Starting Point

The objective is to derive an open-system specification of the mechanism that controls elevators and interacts with passengers to transport them to their destination floors (henceforth this mechanism is referred to as the controller). The starting point is a model of the composite system of which the controller is a part; this takes the form of a closed-system specification. Since in principle the system encompasses the physical universe, what is actually modeled is a manageable abstraction that retains only relevant details.

5.1.1 *System and Agents.* The multistory building is modeled very simply as a sequence of floors; elevators are modeled as objects with a location (a floor) and

doors (either open or closed); passengers are modeled as objects with a destination (a floor) and a location (a floor or elevator). Passengers are modeled only during their interaction with the elevators, so the appearance of a passenger who wishes to use the elevators to get from one floor to another is modeled by the creation of a new passenger object with appropriate location and destination. This simple model is divided into the following agents:

—Each passenger is a separate agent (in contrast to, say, using a single agent for all passengers), so as to be able to consider what an individual passenger must know and do.
—The controller is modeled as a single agent (which abstracts from details of how an actual control mechanism interacts with elevator hardware—motors, sensors, switches, etc.).
—The appearance of passengers is an agent that creates passengers (objects that are also agents).

The somewhat counterintuitive introduction of an "appearance" agent derives from the need to separate the activity of passengers who are interacting with the elevators and the activity of passengers outside of such interaction (the latter has been abstracted to bare essentials, i.e., that new passengers appear at floors with other floors as their destinations).

5.1.2 *System Decomposition.* The decomposition of the elevator system into agents determines the information belonging to each agent, the activities done by each agent, and the restrictions on interactions among agents. For example, a passenger's location and destination are information belonging to that passenger agent. When a passenger enters an elevator, that activity is done by that passenger agent.

The extent to which passenger agents may interact with one another and with the controller is determined by their interfaces. By default, no interaction is allowed. Clearly this will not be acceptable. For the purposes of this presentation, consideration of what *are* acceptable interfaces is delayed until after the assigned constraints have been divided among the individual agents.

5.1.3 *System Behavior.* The desired system behavior is the rapid transportation of passengers to their destinations. This is specified by generating the set of all possible transportation histories, and pruning to just those histories in which passengers are transported rapidly to their destination floors.

Possible transportation histories are generated by combining the possible activities of the various agents in the specification (the controller causing elevators to move and open/close their doors, the appearance agent introducing new passengers at floors with other floors as their destinations, and individual passengers entering/exiting elevators). Constraints may be used in defining possible transportation histories (e.g., to express that a passenger at a floor can only enter an elevator whose doors are open at that floor).

Pruning to only rapid transportation histories is defined by constraints (e.g., to express that a passenger must never move further from his/her destination floor). These constraints are assigned as the joint responsibility of the controller and the passengers, but *not* of the appearance agent (for otherwise the behavior

could be trivially achieved by having passengers appear with their destinations always equal to their locations). If there are properties of passenger appearance that it would be advantageous to know (e.g., the majority of passenger travel is to or from the ground floor), they would be expressed as constraints on the appearance agent.

It considerably simplifies the definition of the constraints for rapid transportation if they have only to prune from the set of *possible* transportation histories. That is, rapid transportation constraints *rely* upon the constraints defining possible transportation histories to have already been applied. This is achieved by pruning in two main stages, as shown in Figure 9. This is an example of the use of multistage pruning referred to in Section 4.4.

5.2 Development to Divide Responsibility Among Individual Agents

The development goal is to decompose all constraints with responsibilities spanning multiple agents into constraints with responsibilities separately assigned to the controller and individual passengers.

Decomposition of a constraint is achieved by choosing an implication of the constraint to make into a separate, explicit constraint, and thereafter simplifying the original constraint. Judicious choice of the implication will give a constraint whose responsibility can be simplified to an individual agent. This process is continued until all the constraints are assigned as the responsibility of individual agents.

The development that follows has been done *by hand only*, that is, has *not* been carried out in any formal system that guarantees the correctness of the steps that are intended to preserve behavior. All the stages of the development are expressible in the extended Gist that has been outlined. The stages are as follows:

(1) The initial constraints defining suitably rapid transportation are as follows:

   (a) NO FURTHER FROM DESTINATION. A passenger must never move further from his/her destination floor.
   (b) NO DELAY TO RIDERS. Passengers riding inside elevators must not be unnecessarily delayed. "Unnecessary delay" can be defined on a history as a contiguous sequence of states during which a passenger was inside the elevator while the elevator remained inactive (did not move, open or close its doors, or take on or let off passengers).

   These constraints are initially assigned as the joint responsibility of the controller and all passengers.

(2) Decompose NO FURTHER FROM DESTINATION by:

   (a) Defining the (single-valued) Passenger Direction (P-D) of a passenger to be the direction (up or down) in which that passenger must go to reach his/her destination floor. (More precisely, the P-D of a passenger will have no value when the passenger is at his/her destination floor, so it is either single-valued or has no value.) *This definitional step names a piece of information in preparation for future steps.*

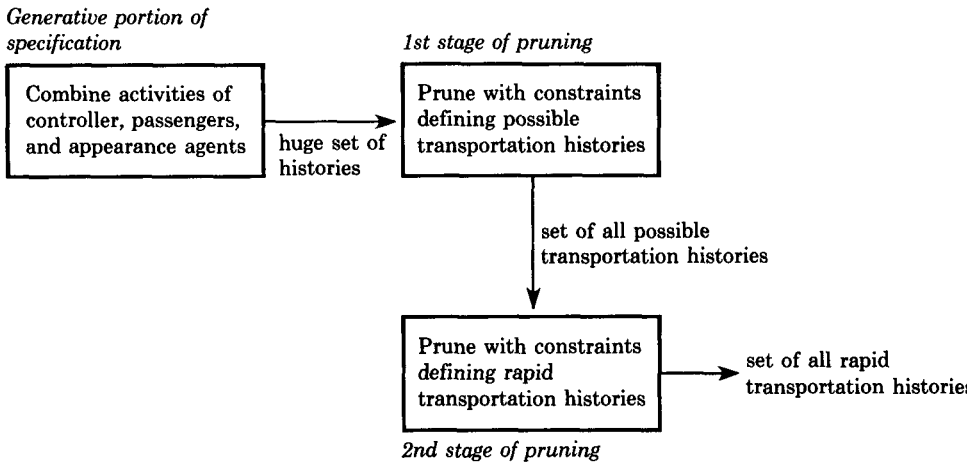


Fig. 9. Main stages of pruning in the initial elevator specification.

(b) Choosing the implication of NO FURTHER FROM DESTINATION, that all riders in a moving elevator have the same P-D (or none at all), to become the explicit constraint RIDERS IN MOVING ELEVATOR COMPATIBLE. This is assigned as the responsibility of the controller and all passengers.

(c) Taking advantage of the introduced constraint to simplify NO FURTHER FROM DESTINATION; its simplified form is that a moving elevator with a rider must be moving in that rider's P-D direction, and is the responsibility of only the controller.[5] The constraint is renamed accordingly to MOVE IN RIDER'S P-D. *Thus the introduced constraint allowed* NO FURTHER FROM DESTINATION *to be simplified to a responsibility of the controller alone.*

Figure 10 illustrates the development up to and including step (2c). The double-bordered boxes show the constraints and their associated sets of responsible agents that are in effect following this stage of development, while the single-bordered boxes show the constraints that have been decomposed (at this stage

[5] This simplification is intended to leave unchanged the behavior denoted by the specification. NO FURTHER FROM DESTINATION is equivalent to the conjunction of RIDERS IN MOVING ELEVATOR COMPATIBLE and MOVE IN RIDER'S P-D; the questionable step is the assignment of the singleton set of only the controller as the set of agents responsible for this latter constraint. The danger in this is that lopping a transition in which an elevator with a rider moves the wrong way limits choices of not only the controller, but also of some passenger(s), and whereas previously the passengers shared responsibility, now it rests on the controller alone, thus not admitting the same pruning. To see that this does not arise, compare any transition in which the elevator moves the wrong way with the similar transition comprising all the same changes but for the elevator moving the right way. If moving the right way succeeds, then lopping the transition of moving the wrong way limits only the controller's choice of which way to move the elevator (since the same choices of all the other agents are present in the retained transition). Conversely, if moving the right way fails, then it is intuitively clear that whatever constraints caused its failure must also apply when moving the wrong way, and hence if pruning of the right way transition is valid, then pruning of the wrong way transition must also be valid.

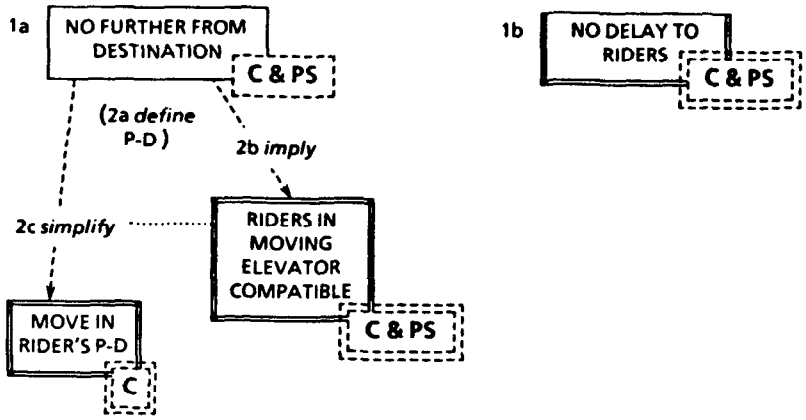


Fig. 10. Development up to and including step (2c).

there is only one such constraint). Stages of decomposition (steps (2b) and (2c)) are indicated as labelled arrows; the label "imply" on (2b) signifies that the constraint at the source of the arrow (NO FURTHER FROM DESTINATION) implies the constraint at the target, while the label "simplify" on (2c) indicates that the constraint at the source of its arrow is simplified into the constraint at its end by taking advantage of the other constraint pointed to by the lightly dotted line (RIDERS IN MOVING ELEVATOR COMPATIBLE). Using these same notational conventions, the entire development is shown later in Figure 11.

(3) Decompose NO DELAY TO RIDERS by:

(a) Choosing the implication that all riders in a moving elevator have the same P-D to become the explicit constraint RIDERS IN ELEVATOR COMPATIBLE. This is assigned as the responsibility of the controller and all passengers.

(b) Introducing constraint EXIT WHEN AT DESTINATION, that a passenger in an elevator exits that elevator when *and only when* it is at his/her destination floor. This is assigned as the responsibility of the controller[6] and all passengers. *This is* not *implied by the existing constraints, since it eliminates otherwise acceptable histories in which a passenger exits prior to reaching his/her destination.*[7] *Hence this step* modifies *the specification, rather than simply transforming its form while leaving the denoted behavior unchanged. The motivation for this modification is to simplify the behavior of passengers riding elevators. This will make it*

[6] The controller's inclusion is necessary to have the controller open elevator doors.

[7] Under certain circumstances the eliminated transportation history may even have been preferred from a global viewpoint. For example, suppose the last remaining passenger inside an elevator exits prior to reaching his/her destination, thus freeing that elevator to reverse direction and to pick up a multitude of passengers wishing to go the opposite way (i.e., delaying the one passenger in order to better serve the many). Irrespective of the desirability or otherwise of such a history, it is clear that to coordinate the passengers and controller to achieve this would require more interagent communication than is commonly found in real-world elevator systems.

easier to isolate what responsibility the controller has for the NO DELAY TO RIDERS *constraint. The preceding step* (3a) *made explicit the consistency condition on passengers within the same elevator; this modification makes explicit* (*and simplifies*) *the condition on passenger exit from elevators.*

(c) Simplifying NO DELAY TO RIDERS. To do this successfully requires the use of information and constraints that will be defined during decomposition of other constraints. This simplification step is delayed accordingly.

(4) Decompose EXIT WHEN AT DESTINATION by:

(a) Choosing the implication that an elevator at the destination floor of some rider in that elevator must open its doors to become the explicit constraint OPEN DOORS FOR EXITOR. This is initially assigned as the reponsibility of the controller alone.

(b) Taking advantage of the introduced constraint to simplify the assignment of responsibility for EXIT WHEN AT DESTINATION from the controller and all passengers to only the passenger who is the exiting rider.

(5) Since RIDERS IN ELEVATOR COMPATIBLE implies RIDERS IN MOVING ELEVATOR COMPATIBLE, the latter has been subsumed and so is discarded.

(6) Associate with each elevator an Elevator Direction (E-D), which is nondeterministically varying, subject to the following constraints:

—E-D is a direction (responsibility assigned to the controller);
—E-D is single-valued (responsibility assigned to the controller);
—MOVE IN E-D, the constraint that when an elevator moves, its E-D = the direction of movement (responsibility assigned to the controller); and
—E-D = RIDER'S P-D, the constraint that when an elevator has a passenger inside, its E-D = that passenger's P-D (responsibility assigned to the controller and all passengers).

For every elevator, there is always some E-D value satisfying the above constraints,[8] *hence this definitional step makes no change to the behavior, other than adding some nondeterminism of varying E-D values when they are not uniquely determined. E-D will be useful in reexpressing some of the existing constraints and for communicating between the controller and the passengers.*

(7) Since both E-D = RIDER'S P-D and MOVE IN E-D imply MOVE IN RIDER'S P-D, the latter has been subsumed and so is discarded.

(8) Since both E-D = RIDER'S P-D and the single-valuedness of E-D imply RIDERS IN ELEVATOR COMPATIBLE, the latter has been subsumed, and so is discarded.

[8] Occupied elevators take as E-D value the P-D of their rider(s), which is unique because of RIDERS IN ELEVATOR COMPATIBLE; moving elevators take as E-D value their direction of movement; MOVE IN RIDER'S P-D ensures consistency for elevators that are simultaneously moving and occupied.

(9) Decompose E-D = RIDER'S P-D by the following:

(a) Choosing the implication that an elevator's E-D must remain constant while there is a passenger inside to become the explicit constraint E-D CONSTANT WHILE RIDDEN. This is assigned as the responsibility of the controller.

(b) Introducing the constraint NO SIMULTANEOUS ENTRY AND E-D CHANGE, that a passenger may not enter an elevator simultaneous with that elevator's E-D value changing. This is assigned as the responsibility of the controller and all passengers. *This eliminates some changing of E-D values, but leaves all other choices (of passenger entry, etc.) unaffected, hence makes no change to the behavior other than limiting some of the nondeterminism introduced in step* (6). *The motivation for this modification is to simplify the correlation between E-D and riding passenger's P-Ds. The preceding step* (9-a) *made explicit the need to keep the E-D constant while an elevator is being ridden; this modification simplifies the situation in which a passenger boards an elevator, by removing the tricky case of the E-D changing in parallel with the passenger boarding.*

(c) Introducing the constraint DOORS CLOSED WHILE E-D CHANGES, that an elevator's doors must be closed while its E-D value changes. This is assigned as the responsibility of the controller alone. *This reduces some last minute changing of E-D values, but leaves all other choices (of door closing, etc) unaffected, hence makes no change to the behavior other than further limiting some of the nondeterminism introduced in step (6). The motivation is to simplify the conditions under which the controller may change E-D values so as to be able to isolate what responsibility a passenger boarding an elevator has for the* NO SIMULTANEOUS ENTRY AND E-D CHANGE *constraint.*

(d) Since DOORS CLOSED WHILE E-D CHANGES implies NO SIMULTANEOUS ENTRY & E-D CHANGE, the latter has been subsumed and so is discarded.

(e) Taking advantage of the constraints introduced in the preceding steps to simplify E-D = RIDER'S P-D; its simplified form is that a passenger entering an elevator must have a P-D value equal to the elevator's E-D value, and is the responsibility of the entering passenger only. The simplified constraint is renamed accordingly to ENTER ONLY IF E-D = P-D.

(10) Last, return to the decomposition of NO DELAY TO RIDERS begun in step (3). So far, the introduced constraints isolated when a passenger may *not* enter an elevator; now a constraint is introduced to determine when a passenger *must* enter. Having done this, it will be possible to assign the remainder of the responsibility for the original constraint to the controller. This is done by the following:

(a) Introducing a constraint IMMEDIATE ENTRY, that a waiting passenger does not pass up the opportunity to enter an appropriate elevator (i.e., an open-doored elevator at that passenger's floor with E-D = the passenger's P-D; if there are several such elevators, then any one of

them). This is assigned as the responsibility of the entering passenger only. *This constraint is* not *redundant, because it eliminates otherwise acceptable passenger actions. Hence this step, like the introduction of* EXIT WHEN AT DESTINATION (step (3-b)), *is a* modification *to the denoted behavior, in this case one that simplifies passenger entry into elevators.*

(b) Taking advantage of the constraints introduced in step (3) and in the previous step to simplify NO DELAY TO RIDERS; its simplified form is that an occupied elevator must not stop at a floor unless it is the destination of one of the riders, or the location of a waiting passenger who will board (i.e., who has a P-D equal to the elevator's E-D), and is the responsibility of the controller only. The simplified constraint is renamed accordingly to STOP ONLY TO ALLOW ENTRY/EXIT.

5.3 End Point

The entire development is sketched in Figure 11.

Each of the remaining constraints is now the responsibility of a single component (controller or passenger). Some intertwining between specification and implementation occurred, as evidenced by the steps (3-b) and (10-a) where constraints were introduced that modified the denotation. The definition of the latter modification (introduction of the IMMEDIATE ENTRY constraint) relied upon concepts introduced in the course of the development (E-D and P-D), and hence would have been particularly hard to formulate as part of the initial specification.

5.3.1 *Agent Interfaces.* Until now, no consideration has been given to what each agent's interface should be, that is, what access an agent has to information belonging to other agents. In truth, these considerations have already influenced the development, for example, part of the motivation for introducing the E-D value for elevators was to serve as a communication between the controller and individual passengers.

Information belonging to the controller is the locations of elevators, their E-D values, and the status of their doors.

Information belonging to an individual passenger is that passenger's destination and location (if inside an elevator, then that elevator; if outside, then the floor).

The following interfaces provide access to sufficient information to permit implementation.

—Allow the appearance agent to create new passengers and initialize their locations and destinations.
—Allow the controller to observe the following: the presence of and destination floors of passengers inside elevators, and the presence of and direction towards destinations of passengers waiting for elevators.
—Allow a passenger to observe the following: when waiting at a floor, the presence of and E-D values of open-doored elevators at that floor, and when riding inside an elevator, the location (floor) of that elevator.

The passenger interface implies some extra activity of the controller to display E-D values and elevator locations to passengers; this corresponds to actual

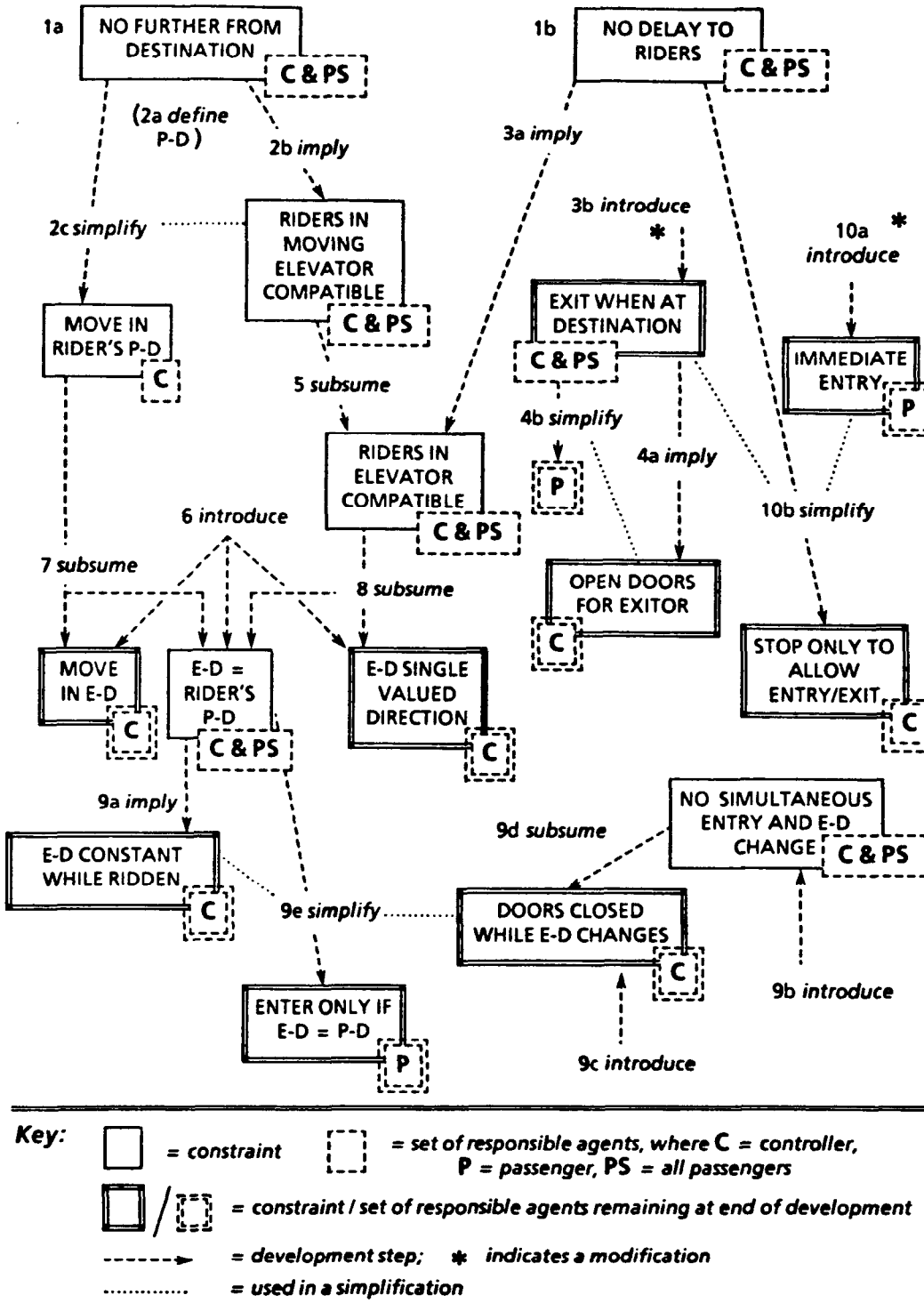


Fig. 11. Elevator scenario development.

real-world elevator systems. Similarly, the controller interface implies some extra activity of passengers to display their directions when waiting for elevators, and to display their destinations when inside elevators; unfortunately, this is not realistic. Rather, passenger behavior is extended to provide information from which the above may be deduced. In practice, passengers send signals to the controller (by pressing buttons at floors or inside elevators), and controller behavior is extended to confirm receipt of such information (by lighting pressed buttons, etc.).

When appearing at a floor, a passenger presses a button corresponding to his/her desired direction (unless that button is lit to show it has already been pressed). Similarly, when entering an elevator, a passenger presses a button corresponding to his/her destination (again, unless that button is already lit). The controller lights those buttons in response to their having been pressed, and unlights them when the condition no longer holds (e.g., when an "up" elevator has stopped and opened its doors at a floor, all the passengers waiting to go up will board, so the controller unlights the "up" button). From these signals, and knowledge of passenger behavior,[9] the controller is able to deduce sufficient information for its correct implementation.

A rather different interface that allowed more access to information between controller and passengers might offer radically different possibilities for the development. For example, suppose all passengers in the building were given communication devices that they used to inform the controller of their presence and desired destinations, and that the controller, through these devices, instructed passengers when and which elevators to enter/exit. Under these circumstances the controller could do more to optimize the global system performance by occasionally delaying a few passengers, as discussed in the footnote to development step (3-b).

Clearly, there is significant intertwining between the interface and behavioral aspects of this specification, which has only been hinted at in this presentation.

5.3.2 *Insights into the Development.* The ordering of the presented development's steps does not necessarily reflect the chronological trace of its construction. Also, blind alleys in the original construction process have been omitted. I will not attempt to relate the entire history, but will simply identify some of the insights that guided construction. Their influence can be discerned on the pedagogical development that has been presented.

One key insight is that transitions inducing interaction among agents are focal points in the development. In our elevator scenario, these transitions are entry and exit of passengers, and elevator movement and door activity. Decomposition of constraints that span multiple transitions and agents is usually aimed at formulating smaller constraints on these focal point transitions. For example, EXIT WHEN AT DESTINATION and IMMEDIATE ENTRY constrain the focal transitions of passenger entry; their presence allows decomposition of NO DELAY TO RIDERS into STOP ONLY TO ALLOW ENTRY/EXIT and RIDERS IN ELEVATOR

[9] For example, knowing that a passenger will exit an open-doored elevator when and only when its doors are open at that passenger's destination floor allows the controller to deduce when passengers exit without requiring passengers to signal their exit in any manner.

COMPATIBLE, the former is a constraint on a focal point of elevator activity, and the latter is further decomposed.

Introducing E-D (Elevator Direction) values as an explicit form of communication between controller and passengers is crucial to coordinating these multiple agents. Clearly, the knowledge that this is a solution applied in real elevator systems must have influenced my construction of the development. However, it is interesting to speculate that I might have been able to invent this by considering the information implicitly shared by the MOVE IN RIDER'S P-D and RIDERS IN ELEVATOR COMPATIBLE constraints.

Recognizing the opportunity to share the RIDERS IN ELEVATOR COMPATIBLE constraint between decomposition of NO FURTHER FROM DESTINATION and NO DELAY TO RIDERS is crucial in eliminating redundancy in the development. Such opportunistic effects are likely to arise in many developments, and since they are difficult to predict in advance, we must expect to have to look over a growing development in order to identify them, and be prepared to reorganize to take advantage of them.

5.3.3 *Completing the Implementation of the Controller.* Having isolated the responsibility for each constraint to individual agents and having introduced all the necessary explicit signalling that each agent must make to other agents, the specification of the controller is now in open form; that is, it makes no use of closed-system concepts, although it may continue to make liberal use of other Gist specification constructs, including constraints. The constraints on the passengers and passenger appearance agent, together with the generative description of how those agents behave, define the complete range of inputs to which the controller must respond. The constraints on the controller, together with its generative description, define the behavior of the controller in response to its inputs.

The derivation of an implementation of the controller, from this open-system specification into a program expressed in some target programming language, falls into the category of conventional program transformation. For example, removing the use of a constraint on the controller is equivalent to implementing nondeterminism and backtracking, or, under favorable circumstances, finding a deterministic algorithm that will pick correct choices the first time. The general problem of transforming open-system specifications (which may still be far removed from efficient implementations) is quite hard, but falls outside the scope of this paper. The reader interested in how we approach the transformation of Gist specifications into implementations is referred to [1] and [11] for discussions and illustrations of our techniques.

5.3.4 *Development Structure.* This example also illustrates the rich structure of the development from specification to implementation. In the domain of development done by program transformation, there is a clear trend toward richer languages for recording developments and more sophisticated mechanisms for applying them. See [15] for a survey of earlier research in this direction and for a description of Wile's POPART system, a mechanism for (among other things) applying developments expressed in his development language PADDLE. We have applied POPART to the development of some Gist specifications, although

not on the scale of this elevator scenario. Fickas has gone so far as to view both development and specification in the framework of a more general problem-solving process [6]. Presumably we should also record and support the development from closed-system to open-system specifications in a more formal manner than simply the hand-sketched presentation in this paper.

6. CONCLUSIONS

6.1 Related Work

Other researchers have recognized the above motivations and have recommended their equivalent of what I call the closed-system style of specification. The surveys below provide many pointers to related research.

6.1.1 *Surveys of Related Work.* The Executable Metric Models Applications (EMMA) project, at Imperial College of Science and Technology, London, was initiated as the first step toward developing methods, techniques, and tools to support the entire software development process, including system evolution. Their final report [4] presents their objectives and findings, including a summary of existing and envisaged tools, techniques, and languages for modeling systems prior to implementation. They argue for "a requirement representation which specifies the system and its environment as a closed world. This 'embedded' style of requirement is contrasted with a 'black box' style, in which the system boundary is explicit, and the environment is modeled only as a source of inputs and sink for outputs." They observe that approaches based on the rigorous application of correctness-preserving transformations (of which ours is an instance) are "techniques which have yet to be tested for significant applications." Of the development methods in common use, the only one they find to match their desired software development process is the Jackson System Development Method, JSD [10]. JSD is based on *informal* specifications, and thus does not have automated support for development. To its credit, however, it is in real-world use.

Another survey is Zave's description of the "operational" approach to software development [17]. Zave characterizes such an approach as one that uses an executable model of the proposed system interacting with its environment as the starting point from which to derive the implementation. Cited by Zave as instances of the operational approach are JSD (as in the EMMA report), Applicative Programming, our own project, and Zave's PAISLey project. Zave reminds us that many of the ideas of this approach are relatively new and untested, and warns that they may not deliver in practice what they deliver in theory; on an optimistic note, she says of these ideas: "They are interesting in their own right for the new perspective they provide, and if successful will yield substantial gains in software productivity."

6.1.2 *Specific Related Research.* Another ongoing research effort based upon formal specification is Zave's own PAISLey project, described in detail in [16]. Zave separately specifies system behavior and system decomposition. PAISLey is Zave's language for specifying system behavior. PAISLey specifications take the form of a collection of processes, each of which represents one of the

autonomous entities in the domain being specified. The behavior of a process is a sequence of states, described by an applicative language definition of a successor function. Interactions between processes take place via "exchange functions," which carry out the side effect of asynchronous interaction, but which, within a process itself, appear as normal functions. This is very different from Gist, which takes a much more global view of the whole system in its approach to specifying behaviors. Gist's ability to use information drawn from the whole system in descriptions of both behavior and decomposition appears to have no direct analogue within PAISLey.

Our research ambitiously aims to assist a wide range of software development. Narrowing one's goals somewhat may permit a more immediate realization of techniques for some specific problem domains. For example, Merlin and Bochmann [12] study a method to elaborate the specification of the "submodules" (components) of a system; if the system consists of a collection of submodules, and the system and all but one of its submodules are specified, then their method elaborates the specification of that remaining module. By restricting their attention to specifications given in terms of sets of possible execution sequences, they emerge with a formula for the remaining module's specification. In the further restricted context of finite state machines, they have a constructive algorithm evaluation of the formula. They see this approach as being useful in the design of distributed systems in general, and find a particular application in the design of communication protocols.

Milner's CCS (Calculus of Communicating Systems) [13] and Hoare's CSP (Communicating Sequential Processes) [9] are well-known approaches to studying and specifying the behavior of ongoing processes. The similarity of our tree representations of behavior to Milner's CCS trees has already been identified (Section 4.2.1). However, in marked contrast to our proposed approach, both CCS and CSP favor the construction of systems by the combination of subsystems whose only communication with one another is through explicit ports. The behavior of such a system is derived from the combination of the behaviors of its subsystems. Their disciplined construction techniques permit the well-structured expression of composite systems. We have chosen instead to specify the behavior of a composite system directly, making liberal use of constructs that draw information from across the entire system without regard to component boundaries. Our preference is to derive the behaviors of the individual components from such a specification, and our emphasis is to seek structure primarily in the development process itself rather than in the programs that emerge from that process.

6.2 Summary

The premise of this paper has been that a composite system's components should be implemented by developing them from a specification of the behavior required of that system and a specification of how that system is divided into components. This implies the need for a specification language to express the various stages in developments from composite systems to implementations of their components.

Our group's specification language, Gist, appears suited to this purpose. Of Gist's existing features, the following combine to be of particular utility.

—Gist is an operational specification language. The denotation of a Gist specification of a system is the set of acceptable histories that that system may exhibit. This is propitious for the specification of system behaviors that are ongoing and/or nondeterministic.
—A "generate and prune" paradigm is used to determine denotation, encouraging a simple specification style in which a broad set of candidate histories is generated, clearly encompassing all the desired histories, after which pruning removes those histories not meeting the easily stated requirements.
—Language constructs used in defining both the generation and pruning of the set of candidate histories have liberal access to information within those histories. It is important to stress that the specification of a composite system's behavior and its decomposition into components can each be expressed in terms of information from throughout that system. Only the components' implementations must abide by the restrictions of the decomposition.

The following enhancements of Gist have been proposed for further supporting composite-system specification and development.

—Gist's agents model the components of a system. Agents partition the generative portion of a specification so that each activity can be associated with the agent that performed it. Additionally, information may belong to agents. This permits modeling restrictions on the extent to which one component may access or affect information belonging to another component.
—Formal meaning is given to the notions of "choice" and "responsibility" in terms of Gist denotations. These notions appear to play a major role in describing the stages of decomposition of system behavior into individual component behaviors.

Further work is required to consolidate some of these specification ideas. For example, accommodating the notion of *changing* the set of agents responsible for a constraint during the course of a history appears possible. Extensive work remains to be done to provide mechanized support for the development of implementations from closed-system specifications.

APPENDIX. Outline of Proof that OKPrunings is Closed under Union

THEOREM. (*From Section* 4.2.4.) *The set of behaviors returned by OKPrunings is closed under union, that is,*

$$\forall b, sac, r1 \in \text{OKPrunings}(b, sac), \\ r2 \in \text{OKPrunings}(b, sac) \mid (r1 \cup r2) \in OKPrunings(b, sac).$$

Its proof is provided as an illustration of the formal manipulations that are necessary to reason about behaviors and their pruning.

PROOF. Unfolding the definition of OKPrunings and labeling the conjuncts, the objective is to show that

[1] $r1 \subseteq b \wedge$
[2] OKPruning(b, $r1$, sac) $\wedge$
[3] $(\forall h \in r1, (sa \times c) \in sac \mid c(h))$
[4] $r2 \subseteq b \wedge$
[5] OKPruning(b, $r2$, sac) $\wedge$
[6] $(\forall h \in r2, (sa \times c) \in sac \mid c(h))$

implies

[7] $(r1 \cup r2) \subseteq b \wedge$
[8] OKPruning(b, $(r1 \cup r2)$, sac) $\wedge$
[9] $(\forall h \in (r1 \cup r2), (sa \times c) \in sac \mid c(h))$

[1] and [4] imply [7] (trivially), and [3] and [6] imply [9] (also trivially).

Unfolding OKPruning in [2], [5], and [8], it remains to show that

[10] $\forall n \in r1 \mid \forall t \in \text{LAN}(n, b, r1) \mid \text{LBL}(t, n, r1) \subseteq \text{RFL}(t, b, sac) \wedge$
[11] $\forall n \in r2 \mid \forall t \in \text{LAN}(n, b, r2) \mid \text{LBL}(t, n, r2) \subseteq \text{RFL}(t, b, sac)$

imply

[12] $\forall n \in (r1 \cup r2) \mid \forall t \in \text{LAN}(n, b, (r1 \cup r2)) \mid \text{LBL}(t, n, (r1 \cup r2)) \subseteq \text{RFL}(t, b, sac)$

abbreviating LoppedAtNode as LAN, LimitedByLopping as LBL, and ResponsibleForLopping as RFL.

For any $n \in (r1 \cup r2)$, we may assume without loss of generality that $n \in r1$, hence from [10],

$$\forall t \in \text{LAN}(n, b, r1) \mid \text{LBL}(t, n, r1) \subseteq \text{RFL}(t, b, sac)$$

$r1 \subseteq (r1 \cup r2)$, so by Lemma 1,

$$\text{LAN}(n, b, r1 \cup r2) \subseteq \text{LAN}(n, b, r1),$$

hence

$$\forall t \in \text{LAN}(n, b, r1 \cup r2) \mid \text{LBL}(t, n, r1) \subseteq \text{RFL}(t, b, sac)$$

$r1 \subseteq (r1 \cup r2)$, so by Lemma 2,

$$\text{LBL}(t, n, r1 \cup r2) \subseteq \text{LBL}(t, n, r1)$$

hence by transitivity of $\subseteq$,

$$\forall t \in \text{LAN}(n, b, r1 \cup r2) \mid \text{LBL}(t, n, r1 \cup r2) \subseteq \text{RFL}(t, b, sac).$$

Since this holds for any $n \in (r1 \cup r2)$, this proves [12] □

LEMMA 1. *$r \subseteq rr$ implies* $\text{LAN}(n, b, rr) \subseteq \text{LAN}(n, b, r)$. *Intuitively, retaining more histories reduces lopping at nodes.*

PROOF.

$\text{LAN}(n, b, rr) = \text{TFN}(n, b) - \text{TFN}(n, rr)$,
abbreviating TransitionsFromNode as TFN

$r \subseteq rr$, so $\text{TFN}(n, r) \subseteq \text{TFN}(n, rr)$ (trivial), hence
$\text{LAN}(n, b, rr) \subseteq \text{TFN}(n, b) - \text{TFN}(n, r) = \text{LAN}(n, b, r)$ □

LEMMA 2. *$r \subseteq rr$ implies* $\mathrm{LBL}(t, n, rr) \subseteq \mathrm{LBL}(t, n, r)$. *Intuitively, retaining more histories decreases the set of agents whose choices are limited.*

PROOF.

$$\mathrm{LBL}(t, n, rr) = \{a \mid \{d \mid (a \times \delta) \in t\} \notin \mathrm{CAN}(n, rr, b)\},$$
abbreviating ChoicesAtNode as CAN

$r \subseteq rr$, so by Lemma 3,

$$\mathrm{CAN}(n, rr, b) \supseteq \mathrm{CAN}(n, r, b),$$

hence

$$\mathrm{LBL}(t, n, rr) \subseteq \{a \mid \{d \mid (a \times \delta) \in t\} \notin \mathrm{CAN}(n, r, b)\} = \mathrm{LBL}(t, n, r) \quad \square$$

LEMMA 3. *$r \subseteq rr$ implies* $\mathrm{CAN}(n, rr, b) \supseteq \mathrm{CAN}(n, r, b)$. *Intuitively, retaining more histories increases agents' choices.*

PROOF.

$$\mathrm{CAN}(n, rr, b) = \{\{d \mid (\delta \times a) \in t\} \mid t \in \mathrm{TFN}(n, rr)\}$$
$r \subseteq rr$, so $\mathrm{TFN}(n, rr) \supseteq \mathrm{TFN}(n, r)$ (trivial),

hence

$$\mathrm{CAN}(n, rr, b) \supseteq \{\{d \mid (a \times \delta) \in t\} \mid t \in \mathrm{TFN}(n, rr)\} = \mathrm{CAN}(n, r, b) \quad \square$$

ACKNOWLEDGMENTS

Present and former members of Bob Balzer's specification group at ISI have collectively defined the rich context within which this work lies. Particular thanks are due to Don Cohen, Neil Goldman, Jack Mostow, and Dave Wile for relevant discussions and constructive comments on drafts of this paper. Meetings of IFIP Working Group 2.1 have provided valued inspiration and feedback. Thanks are also due to Sheila Coyazo for scrutinizing an earlier draft, and to the referees for their suggestions.

REFERENCES

1. BALZER, R. Transformational implementation: An example. *IEEE Trans. Softw. Eng. SE-7*, 1 (1981), 3–14.
2. BALZER, R., AND GOLDMAN, N. Principles of good software specification and their implications for specification languages. In *Specification of Reliable Software*. IEEE Computer Society, 1979, 58–67.
3. BALZER, R., GOLDMAN, N., AND WILE, D. Operational specification as the basis for rapid prototyping. *ACM Sigsoft Softw. Eng. Not. 7*, 5 (Dec. 1982), 3–16.
4. BARTLETT, A. J., CHERRIE, B. H., LEHMAN, M. M., MACLEAN, R. I., AND POTTS, C. The role of executable metric models in the programming process—*final report*. Tech. Rep., Dept. of Computing and Control, Imperial College, London, 1984.
5. DARLINGTON, J. A synthesis of several algorithms. *Acta Inf. 11*, 1 (Dec. 1978), 1–30.
6. FICKAS, S. F. Automating the transformational development of software. Ph.D. thesis, Univ. of California, Irvine, 1982.
7. GREEN, C., LUCKHAM, D., BALZER, R., CHEATHAM, T., AND RICH, C. Report on a knowledge based software assistant. Tech. Rep. RADC-TR-83-195, Rome Air Development Center, Aug. 1983.
8. HENNESSEY, M., AND MILNER, R. Algebraic laws for nondeterminism and concurrency. *J ACM 32*, 1 (Jan. 1985), 137–161.

9. HOARE, C. A. R. Communicating sequential processes. *Commun. ACM 21*, 8 (Aug. 1978), 666–677.
10. JACKSON, M. A. *System Development.* Prentice-Hall, Englewood Cliffs, N.J., 1983.
11. LONDON, P. E., AND FEATHER, M. S. Implementing specification freedoms. *Sci. Comput. Program. 2* (1982), 91–131.
12. MERLIN, P., BOCHMANN, G. V. On the construction of submodule specifications and communication protocols. *ACM Trans. Program. Lang. Syst. 5*, 1 (Jan. 1983), 1–25.
13. MILNER, R. A calculus of communicating systems. In *Lecture Notes in Computer Science, 92.* Springer-Verlag, New York, 1980.
14. SWARTOUT, W., AND BALZER, R. On the inevitable intertwining of specification and implementation. *Commun. ACM 25*, 7 (July 1982), 438–440.
15. WILE, D. S. Program developments: Formal explanations of implementations. *Commun. ACM 26*, 11 (Nov. 1983), 902–911.
16. ZAVE, P. An operational approach to requirements specification for embedded systems. *IEEE Trans. Softw. Eng. SE-8*, 3 (May 1982), 250–269.
17. ZAVE, P. The operational versus the conventional approach to software development. *Commun. ACM 27*, 2 (Feb. 1984), 104–118.

Received May 1985; revised May 1986; accepted June 1986